1.表达式：

JavaScript 参考 - JavaScript | MDN (mozilla.org)

分为两种，一种是原始表达式，另一种是复杂表达式

1.1原始表达式

1. 直接量：也叫字面量，是指程序中能直接使用的数据值

```
var a = null;
var boolean = true;
var boolean = false;
undefined;
var n = 1;
var str = 'abc';
var obj = {a:123};
var str = 'new Date()';
var b;
```

2. 保留字：尽可能的不用拿来作为标识符用；

```
var let function for if do while 等
```

3. 变量名：遵守命名规则，并做到见名知意

1.2初始化表达式

数组和对象初始化实际上是一个以字面量方式描述的初始化过程。

- 数组初识化：
- 对象初始化：

```
var arr = [];//空数组
var arr =[1,2,3];
var arr =[['a', 1], ['b', 2], ['c', 3]];

var obj = {};//空对象
var obj = {a:1};
```

1.3复杂表达式

- 属性访问表达式

```
//1.表达式后跟随一个句点和标识符，表达式指定对象，标识符则指定需要访问的属性名称，相当于‘的’
//2.使用方括号，方括号内是另一个表达式（这种方法适用于对象和数组），
//方括号内的表达式指定要访问的属性名称或要访问的数组元素的索引
var obj = { a : 1};
    console.log(obj.a);  //1
    console.log(obj['a']); //1

var arr = [1, 2, 3];
    console.log(arr[0]); //1

var arr2 = [1, 2, 3];
    console.log(arr2[2]);//3

var obj = { 'name': '小A', age: 19, gender: 'female' };
    console.log(obj.name); //小A
    console.log(obj.gender); //female
    console.log(obj['gender']); //female 用[]访问属性值，属性要加引号
var key = 'gender';
    console.log(obj[key]);  //female 使用[]对变量进行解析


        /**
         * 深拷贝：拷贝的是数据
         *    js中基本类型数据：都是深拷贝 ，因为基本类型数据都存在 【堆中】
         * 浅拷贝：拷贝的是指针
         *    引用类型数据都是浅拷贝  因为引用类型数据都存在【栈中】
         */
        //  深拷贝
        var a = 23;
        var b = a;
        b = 24;
        console.log(a);
        console.log(b);

        // 浅拷贝
        var o = {
```

```
            name: 'o';
        }
        var p = o;
        p.age = 14;
        console.log(o);
        console.log(p);

        function fun(){
            var x = 0;
            return function(){
                console.log(x);
            }
        }

        function fun2(){
            var a = fun();
            a();
        }
        fun2();
        var res = fun();
        res();

    /**
         * 数组深拷贝思路
         * 1：声明一个空数组；在堆中新存一个数据
         * 2：变量另一个数组；获取到每一个数据码
         * 3：将获取到的数据；添加到空数组中 完成拷贝
         * 因为；arr 与 copyArr 变量；存的是不同的指针；所以改变不受影响
         */
        var arr = [1,3,43,45,6];
        var copyArr = [];
        for(var i =0;i<arr.length;i++){
            copyArr.push(arr[i]);
        }
        copyArr.push('朱明星');
        console.log(arr); // [1, 3, 43, 45, 6]
        console.log(copyArr); //[1, 3, 43, 45, 6, '朱明星']

        // 引用类型数据原理
        // 在堆中多存一个数据；让这个数据取拷贝另一个数据中内容
```

```
        // 导致：每个变量存的是不同的指针
        var obj = {
            name:'人才',
            age:12;
        }
        var copyObj = {};
        for(var x in obj){
            copyObj[x] = obj[x];
        }

        copyObj.newName='八戒';
        console.log(obj); // {name: '人才', age: 12}
        console.log(copyObj);// {name: '人才', age: 12, newName: '八戒'}
```

- 对象创建表达式

```
var obj1 = new Object();
var obj2 = Object.create();
```

- 函数表达式

```
var fn = function(){};
//表达式函数
var fn = function(){
    return 100;
 };
   console.log(fn() + '' + 200);//100200
//fn()调用函数;
// '' 把100转成字符串'100';
// '100'+200-->100200
```

2.运算符

2.1算数运算符：(一元运算符)

运算符	运算	范例	结果
+	正号	+3	3
-	负号	b=4; -b	-4
+	加	5+5	10
-	减	6-4	2
*	乘	3*4	12
/	除	5/5	1
%	取模(取余)	7%5	2
++ ++	自增（前）：先运算后取值 自增（后）：先取值后运算	a=2;b=++a; a=2;b=a++;	a=3;b=3 a=3;b=2
- - - -	自减（前）：先运算后取值 自减（后）：先取值后运算	a=2;b=- -a a=2;b=a- -	a=1;b=1 a=1;b=2
+	字符串连接	“He”+”llo”	“Hello”

```
var n = 20, m = 30;
    console.log(n + m);//50
    console.log(n - m);//-10
    console.log(n * m);//600
    console.log(n / m);//0.6666666666666666
    console.log(n % m);//20

var n = 20, m = 30;
    console.log(++n);//21  先运算，后取值
    console.log(n++);//21  先取值，后运算（返回先取的值）
    console.log(n);//22  返回上一步运算后的值

var a = 3, b = 8, c = 10;
    console.log((++a) - (--c) + (b++) + (++b)); //13
    //4 - 9 + 8 + 10

var k = 1;
// k++整体也有一个值 如果++在后，整体的值是旧值
// 所谓的旧值就是指没有加1的值
var r = k++; // k的值肯定要加1
    console.log(r); //1
    console.log(k); // 2

var j = 100;
var k = ++j;
    console.log(k); // 101
    console.log(j); // 101
```

```
28
29 var n = 5;
30     n++;
31     ++n;
32     console.log(n); //7
33
34 var n = 100;
35 // + 加号   ++自增运算符
36 // 要考虑运算符的优先级
37 // +的优先级是：14  自增运算符的优先级是：17
38 // 自增运算符的优先级 > +的优先级是
39 // 先算++
40 // 10后面加的是++n整体  整体是新值
41 var res = 10 + ++n; // ++n n的值是101  ++n整体的值是101
42     console.log(res); //111
43     console.log(++('52'.split('')[0])); //6 先把'52'分为['5', '2']，在用下标获取5，在先运算，后取值
```

2.2逻辑运算符

1. &&：与

- 与，既要怎样怎样，又要怎样怎样，还要怎样怎样；
- 只要其中一个是false，那么就返回false；
- 只有所有的条件都是true才会返回true；
- 短路机制

2. ||：或

- 有很多条件，你满足其中一个就可以了；你只要怎样怎样就可以了；
- 只要其中一个是true，那么返回值就是true；
- 如果所有的条件都是fase，那么返回false；
- 短路机制；

3. !: 非，取反； 如果本身为true，结果就是false。如果本身为false，结果就是true

- !! 是把一个变量转换成boolean类型；

4. 与或非三者混合到一起使用，组成一个表达式；

```
1 //与
2 var i = 10, j = 20, c=30;
3 var r = (i == 10 && 0 && j == 20 && c == 30);
4     console.log(!!r);//false
5
```

```
//短路机制
var i = 10, j = 20, c=30;
var b = (i == 30 && (c = 60));
    console.log(b);//false, b为false,c=60不再执行，发生短路
    console.log(c);//30

//或
var i = 10, j = 20, c=30;
var r1 = (i == 50 || j == 60 || c == 30 || (c = 50));
    console.log(r1);//true  不再执行
    console.log(c);//30  c==30为true，所以c= 50不再执行，发生短路

console.log(1 && 2 && 3);//3 输出最后面的值
console.log(1||2||3);//1 输出第一个值

//非
console.log(!(i == 30));//true  i==30为false，!取反则为true
```

- &—逻辑与
- |—逻辑或
- ！—逻辑非
- && —短路与
- ||—短路或
- ^ —逻辑异或

a	b	a&b	a&&b	a|b	a||b	!a	a^b
true	true	true	true	true	true	false	false
true	false	false	false	true	true	false	true
false	true	false	false	true	true	true	true
false	false	false	false	false	false	true	false

- 逻辑运算符用于连接布尔型表达式，在Java中不可以写成3<x<6，应该写成x>3 & x<6 。
- “&”和“&&”的区别：

单&时，左边无论真假，右边都进行运算；当符号左边是false时，&继续执行符号

双&时，如果左边为真，右边参与运算，如果左边为假，那么右边不参与运算

- “|”和“||”的区别同理，||表示：当左边为真，右边不参与运算
- 异或(^)与或(|)的不同之处是：当左右都为true时，结果为false。理解：异或，追求的是“异”！

2.3空值合并运算符：

```
    <script>
        // 1
        let height = null;
        let width = null;

        // 重要：使用括号
        let area = (height ?? 100) * (width ?? 50);

        alert(area); // 5000

        // 2
        // || 返回第一个 真 值。
        // ?? 返回第一个 已定义的 值。
        // 换句话说，|| 无法区分 false、0、空字符串 "" 和 null / undefined。它们都一样 —— 假值（falsy values）。
        // 如果其中任何一个是 || 的第一个参数，那么我们将得到第二个参数作为结果。

        // 不过在实际中，我们可能只想在变量的值为 null / undefined 时使用默认值。也就是说，当该值确实未知或未被设置时。
        let height1 = 0;

        alert(height1 || 100); // 100
        alert(height1 ?? 100); // 0

        // 3
        //空值合并运算符 ?? 提供了一种从列表中选择第一个“已定义的”值的简便方式。
        //它被用于为变量分配默认值：

        // 当 height 的值为 null 或 undefined 时，将 height 的值设置为 100
        height = height ?? 100;

        //?? 运算符的优先级非常低，仅略高于 ? 和 =，因此在表达式中使用它时请考虑添加括号。

        //如果没有明确添加括号，不能将其与 || 或 && 一起使用。
    </script>
```

2.4关系运算符

比较运算符： > >= < <= != == === !==等；比较运算符的结果都是boolean型，也就是要么是true，要么是false

==运算符判断相等的流程:

如果值、类型都相同，按照===比较方法进行比较；

如果类型不同，使用如下规则进行比较：

- 如果其中一个值是null，另一个是undefined，它们相等；
- 如果一个值是数字另一个是字符串，将字符串转换为数字进行比较；
- 如有布尔类型，将true转换为1，false转换为0，然后用==规则继续比较；
- 如果一个值是对象，另一个是数字或字符串，将对象转换为原始值然后用==规则继续比较。
- 其他所有情况都认为不相等。

===：值和类型都要相同，恒等

运算符	运算	范例	结果
==	相等于	4==3	false
!=	不等于	4!=3	true
<	小于	4<3	false
>	大于	4>3	true
<=	小于等于	4<=3	false
>=	大于等于	4>=3	true
instanceof	检查是否是类的对象	“Hello” instanceof String	true

```
// a是变量名  变量名是内存空间的别名
var a = 110; // 把110赋值给a所对应的内存空间
    console.log(a); // 对a的操作就是对a所对应的内存空间操作

 var a = 110;
     a = a+666;// a+666 是776,把776赋值给a,把776重新放到a所对应的内存空间中
     console.log(a);//776
     // a = a+666  ==>  a += 666;

var i = 20, j = 20, c = 30;
  if (i == 10) {
    console.log('i的值是10');
  } else if (i < 10) {
    console.log('i的值小于10');
  } else if (i > 10) {
    console.log('i的值大于10');
```

```
  } //i的值大于10

    console.log({} + 'b');//[object Object]b
    console.log({} + 'a');//[object Object]a
    console.log({} + 'b' > {} + 'a');//true

var n1 = new Number(200);
var s1 = new String('200');
var n2 = 200;
    console.log(n1 == n2);//true
    console.log(s1 == n2);//true
    console.log(n1 === n2);//false

// != 使用时不比较类型
    console.log( 1 != "1");  // false
    console.log( 1 != "1abc"); // true
    console.log(1 !== "2"); // true
    console.log(10 !== 20);  // true
```

3.in与 Instanceof:

1. **in:**

- in操作符用来判断某个属性属于某个对象，可以是对象的直接属性，也可以是通过prototype继承的属性(范围包括对象的自有属性和继承属性)
- 和hasOwnProperty的区别一定要知道：不包括原型之上继承过来的；(只检测对象的自有属性)

```
var obj = {a:10, b:200, c: 500};
    console.log(obj);//{a: 10, b: 200, c: 500}

    console.log('a' in obj);//true
    console.log('b' in obj);//true
    console.log('d' in obj);//false obj对象中不存在一个名为'd'的属性

    console.log('toString' in obj);//true  toString在原型之上,obj对象继承了toString()方法
    console.log(obj.hasOwnProperty('toString'));//false

//对于数组属性需要指定数字形式的索引值来表示数组的属性名称（固有属性除外，如length）
var arr = new Array('redwood', 'bay', 'cedar', 'oak', 'maple');
    console.log(0 in arr); // true
```

2. Instanceof:

instanceof运算符希望左操作数是一个对象，右操作数标识对象的类。如果左侧的对象是右侧类的实例，则表达式返回true，否则返回false。JavaScript中对象的类是通过初始化它们的构造函数来实现的。这样的话，instanceof的右操作符应该是一个函数。

```
var obj = {a:10, b:200, c: 500};
    console.log(obj instanceof Object);//true

var arr = [1,23,4];
    console.log(arr instanceof  Array);//true
    console.log(arr instanceof  Object);// true  陷阱

var num = 200;
    console.log(num instanceof Number);//false
    console.log(200 instanceof Number);//false
    console.log(typeof num);//number
    console.log(num.__proto__.constructor === Number);//true

var d = new Date()
d instanceof Date // true d是由Date()创建的
d instanceof Object // true 所有的对象都是Object的实例
d instanceof Number // false d不是一个Number对象

var a = [1, 2, 3]
a instanceof Array // true a是一个数组
a instanceof Object // true 所有数组都是对象


```

底层原理：

为了理解instanceof运算符是如何工作的，必须首先理解“原型链”。原型链作为JavaScript的继承机制。为了计算表达式o instanceof f, JavaScript首先计算f.prototype，然后在原型链中查找o，如果找到，那么o是f (或者f的父类)的一个实例，表达式返回true。如果f.prototype不在o的原型链中的话，那么o就不是f的实例，instanceof返回false。